# Comp 8505

# Selected Topics in Network Security Development

## Assignment One

Dean Morin – A00750619

# Introduction

The purpose of this program is to covertly transmit small amounts of data to and from a compromised machine. The amount being transmitted is small as this is intended as a control channel. Bulk data transfer will have to be handled separately.

# Analysis

Craig Rowland's covert channel application uses one of two methods to discretely transmit one a character at a time:

1. He encodes it into the TCP sequence number or acknowledgement number.
2. He encodes it into the IP identification field.

While both of these solutions are sufficient for most cases, for the truly paranoid there are some issues.

## TCP Sequence/Acknowledgment Number

It could look suspicious to have a stream of ACK or SYN/ACK packets getting a RST in response, all from the same IP address.

## IP Identification Field

RFC 4413 states that the IP identification field can behave one of three ways:

1. Sequential jump
2. Random
3. Sequential

Sequential is complex and rarely used, so I'll ignore it. Sequential jump means that there is a global counter on the machine, and the field increments for every packet sent out. This means that we'd expect the number to go up by one with each packet if the machine only has one communication channel open. With multiple channels we'd expect to see the number jump. This is the technique that I've observed in multiple Linux distros.

Random is just that, but of course they still need to be unique for the particular session. This is used by OSX.

Rowland's implementation could almost pass for random. However, this may look suspicious if it were coming from a Linux box, and also because of the simple encoding technique used, uniqueness isn't guaranteed. If he sent each packet after a

long delay, this wouldn't be a problem, but it would definitely look suspicious if the packets arrived soon after each other. A delay would also be necessary to ensure that the packets don't arrive out of order.

# Design

I'll be using the IP identification field in my implementation. The goal is to address two issues stated earlier:

1. Non-sequential numbers don't look like they originated from a Linux machine
2. Packets can't arrive out of order

To accomplish this, I'll encode the information in the difference between the current ID field and the initial ID field.

This program will send a quick burst of UDP traffic. A jump of 256 could be suspicious so I'll send 4-bits at a time.

As an example, if the initial value of the ID field was 0xA0, then if the next packet had a value of 0xA1 it would encode a 0x0, 0xA2 would encode 0x1 and 0xB0 would encode 0xF.

So that packets don't have to arrive in order, the encoded value will be the initial value + 16 * n + i, where n is the sequence of the packet after the initial packet (1st, 3rd, etc) and i is the encoded value. Some examples would probably help.

| Packet | Acceptable IP ID Range | Received IP ID Field | Encoded Value |
|---|---|---|---|
| 0 | N/A | 0xA0FF | (initial packet) |
| 1 | 0xA100 – 0xA10F | 0xA104 | 0x4 |
| 2 | 0xA110 – 0xA11F | 0xA11F | 0xF |
| 3 | 0xA120 – 0xA12F | 0xA127 | 0x7 |

The initial IP ID will indicate the number of chars being sent, encoded using Rowland's technique of simply not switching to network order. It will also have the 'Don't Fragment' bit set in case it arrives out of order or is lost.

A side effect of this is that it's best to stick with 255 characters or less, otherwise the later ID numbers may exceed the 16-bit maximum for the ID field. For example, if

the message length were 0x01EF, then the initial sequence number would be 0xEF01.

R = max sequence number that we'll use
R = 0x10/datagram * 2 datagrams/character * 0x01EF characters + 0xEF01 (initial ID)

R is greater than 0xFFFF, and the program has no logic to handle this. Some values greater than 255 do work, but the message is intended to be short anyway.

## Signature

The source port number in the transport protocol header will be used to identify a packet that may contain a covert message. The initial packet will also have the 'Don't Fragment' bit set.

For full paranoid mode, it's best to avoid any repetition or pattern. Therefore the signature port will be based on the current date, UTC.

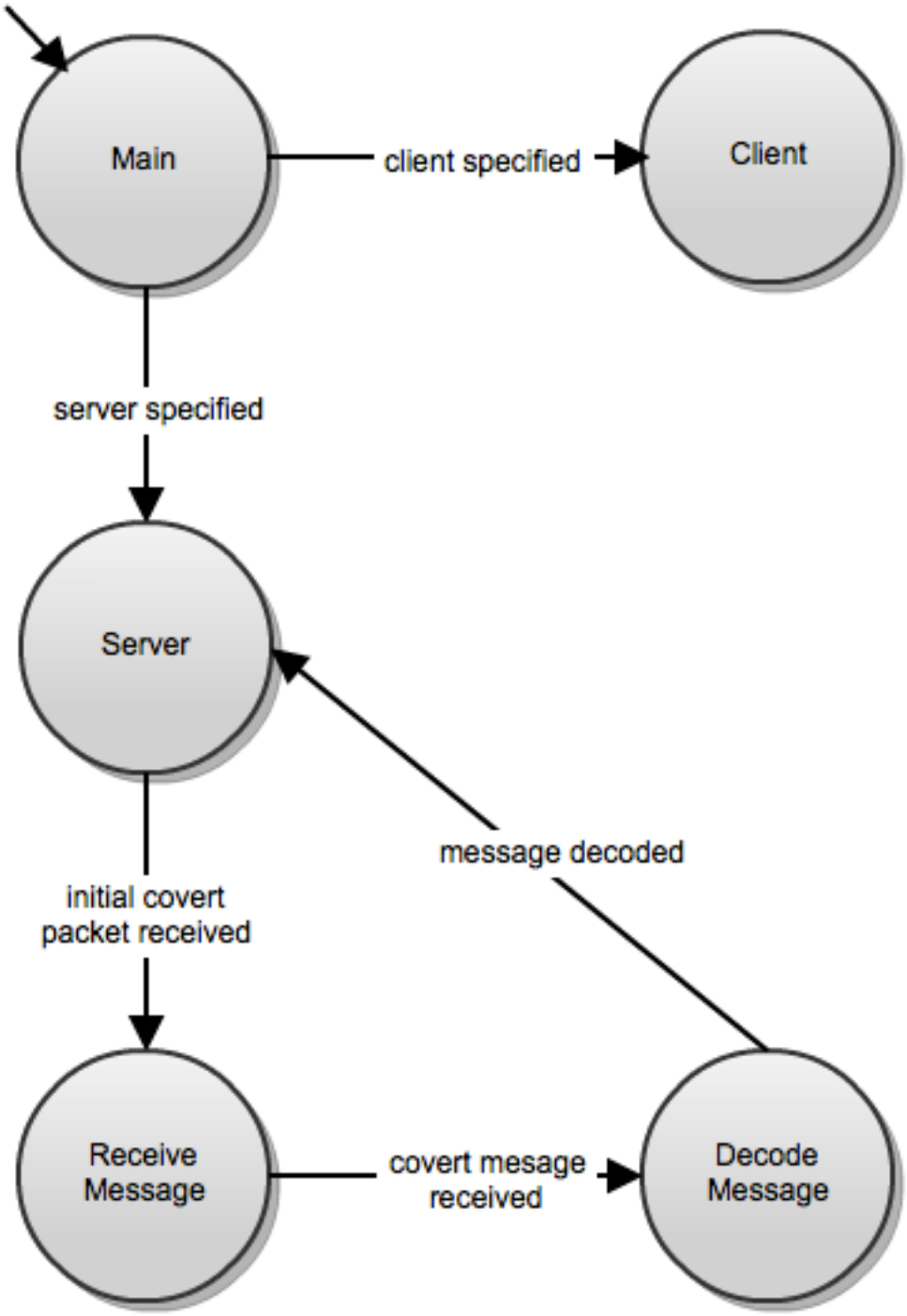
So that this can be later extended to mimic standard web traffic, we should use an ephemeral port as the source port, usually configured to be ports 49151 – 65535.

A simple algorithm can be used to get a port in that range based on the date:

```
port = (50 + month) * 1000
port += (YYYY * day) % 1000
```

Non-covert connections may also use this port, but this won't be a problem since they will be ignored if they don't have the 'Don't Fragment' bit set on only the first packet, or if their IDs don't fall within the acceptable range determined by the initial packet.

# State Transition Diagram



# Psuedocode

Main

- process arguments
- call ‘Client’ or ‘Server’ depending on the arguments

Client

- open the file containing the message
- create a raw socket
- fill the IP header
- fill the UDP header
- fill the UDP pseudo-header
- generate the IP and UDP checksums
- send the initial packet containing the length of the covert message (with the ‘Don’t Fragment’ bit set
- send the covert message, half a character at a time
- close the socket

Server

- create a raw socket and bind it to port 80
- forever loop
    - receive packet
    - if packet is a valid initial packet
        - get length of message
        - ‘Receive Message’

Receive Message

- while read packets < length * 2
    - receive packet
    - store IP ID field value in array

Decode Message

- for each group of 2 ID values in array
    - decode to character and store in decoded character array
- display message and return to forever loop in ‘Server’

# Tests

There were two things that I wanted to know my program handled correctly:

1. Receiving packets out of order.
2. Receiving non-covert packets while a covert message is being received

## Test 1 – Out of Order Packets

This test was to prove that the program can receive covert packets in any order, once the initial packet is received.

### Setup

I rewrote the message sending logic on the client to send all of the even packets first, then all of the odd packets.

### Result

The message was decoded properly on the other end.

## Test 2 – Foreign Packets in the Mix

This test was to prove that the program ignored non-covert packets received on the same port as the covert ones, while a covert transmission was taking place.

### Setup

I ran a simple bash script to spam the server with UDP packets on port 80:

```
#!/bin/bash
while [ 1 ]; do
     nc -zu $1 80
done

$ spam.sh 192.168.0.3
```

I also slowed down the rate of the transmission on the client so that at least a few of the spam packets would arrive before the end of the covert transmission.

### Result

The message was decoded properly on the other end.